Journal of Computer and Knowledge Engineering, Vol. 1, No. 1
DOI: 10.22067/cke.v1i2.58792

Safety Verification of Rate-Monotonic Least-Splitting Real-Time Scheduler on Multiprocessor System

Amin Rezaeian, Abolfazl Ghavidel, Yasser Sedaghat*

Abstract. In real-time task scheduling on multiprocessor systems, partitioning approach has received the attention of many researchers because of its higher least upper bound utilization of safe systems. Semi-partitioning allows some tasks to be split into subtasks and each subtask to be assigned to a different processor. Though task splitting improves the performance of systems, by counting each subtask as a separate task, it increases the effective number of tasks to be scheduled, which in turn, raises the execution overhead. This research is on semi-partitioning of tasks and assigning each partition to a separate processor to be scheduled by the well-known scheduler Rate-Monotonic (RM). Using our algorithm, we do not need to define release time for subtasks of a task to assure their non-concurrent execution and the number of effective tasks, in turn, is reduced. It is theoretically proven that with the proposed semi-partitioning and RM scheduling algorithm, all processors may safely run their tasks according to their deadlines. Further, experimental results on 3000 randomly generated task-sets indicates that not only is utilization factor boosted, but the number of broken tasks also is decreased.

keywords: Rate-Monotonic Least Splitting, Semi-Partitioning, Hard real-time, Multiprocessor Scheduling.

1. Introduction

Recently, the importance of utilizing embedded multicore and multiprocessor system-on-a-chip (MPSoC) has begun to appear as a sensible solution for both power efficiency and high-performance computing in diverse application areas including telecommunications, multimedia systems, computer games, space systems, and process control to name but a few. The importance of the issue becomes more noteworthy when we know that traditional computer-based control systems as well as high technology systems, require high-performance computers for their computations which are usually possible with multiprocessor or multicore systems. A multiprocessor system is composed of several processing elements, called processors, in which all processors can do their processing in parallel. If all processors have the same architecture with one processor repeated design in the processor, the architecture is called homogenous. They all share the same main memory, but each can have their own private cache memory. With this structure, a sequential computation can be shared among many processors if not more than one processor is executing the computation, simultaneously [1]. While manufacturers tend to use multiprocessors in new devices, the development of software facilities which use all available power of multiprocessors [2] is required. In this context, scheduling algorithms play a prominent role in safeness verification of hard real-time systems, i.e., making sure that every request is executed before its deadline. The problem becomes more challenging when there is more than one processor involved so that multiprocessor/multicore systems adds a new dimension to the analysis: how to assign tasks or their requests for different processors. Therefore, there are, overall, two key issues which are still open in multiprocessor scheduling as follows:

1. Task assignment to processors: Finding a target processor to run every selected task;
2. Identifying tasks priority: Making an appropriate order of priority to run tasks.

In this article, the problem of scheduling periodic hard real-time task sets with implicit deadlines on multiprocessors is investigated. What we mean by the implicit deadline is that the deadline of a request is the exact time when the next request arrives from the same task. Many other researchers have studied the same problem but new ideas have kept this practical area dynamic, attractive, and improving.

From the perspective of task migration, task assignment issue on a multiprocessor, generally, fall into two main categories:

1. Global: These algorithms are employed where tasks are allowed to migrate from one processor/core to another.
2. Partitioned: These algorithms are employed where tasks are not allowed to migrate from one processor/core to another.

Moreover, there is another important category which is the hybrid of the above methods called semi-partitioned.

In global scheduling, there is only one queue (or pool) of real-time requests and each processor takes its next execution request from this queue. Regardless of its advantages, in complex and large systems, the overheads of employing a single global queue would become too excessive. In partitioned approach, on the other hand, the set of tasks are divided and each partition is assigned to a separate processor. Finally, in semi-partitioned, some tasks are solely assigned to one processor and some tasks are shared among two or more processors, with the restriction that not more than one processor can work on a request for the shared task, simultaneously. Although the scheduler may be different for each of the three categories of, in almost all cases, the scheduler of all processors is considered to be the same.

Manuscript received September 11, 2016; revised October 17, 2016; accepted November 29, 2016.
Amin Rezaeian, Abolfazl Ghavidel and Yasser Sedaghat,
Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran.
The corresponding author's e-mail is: y_sedaghat@um.ac.ir

While global scheduling techniques, depending on the hardware architecture, potentially could have very high overheads owing to the fact that the task migration from one processor to another usually yields communication and cache missing cost; however, in fully partitioned techniques this does not occur. Nonetheless, non-migration techniques often do not use all available processor capacity. Moreover, it is a widely occurring situation where the total unused capacity may be more than task utilization but no single processor has enough capacity to schedule the task. To solve those problems, a hybrid technique (called semi-partitioned) is used which combines elements of global and partitioned techniques.

It is usually the case that semi-partitioned scheduling leads to a higher overall utilization of the whole system as compared to either of partitioned and global scheduling, for both fixed-priority and dynamic priority. Further, partitioning is a time-consuming task which is computationally equivalent to bin-packing problem that is known to be an NP-hard problem [3]. On the positive side, partitioning is done off-line. Therefore, for a small number of tasks, the time utilized for partitioning is tolerable but, for a large number of tasks efficient heuristics are used. Although an optimal semi-partitioned method has not yet been developed, many heuristic algorithms are presented by researchers. A semi-partitioned approach binds a disjoint set of whole tasks to each processor and allows the remaining tasks to be executed on multiple processors while all shared qualities are defined. In one of the researches on semi-partitioned methods in which Rate-Monotonic (RM) scheduler is used in each processor, worst-case utilization is reported to be $\ln(2) \approx 0.693$ [4].

In this article, a novel semi-partitioned scheduling algorithm called Rate-Monotonic Least Splitting (RMLS) is proposed for multiprocessors. The scheduler of each processor is RM with provisions to avoid simultaneous execution of a shared task by more than one processor. Using this algorithm, we see that the number of split tasks at most is equal to the number of used processors minus one. However, the actual number of split tasks might even be lower. Besides, no task is split into more than two subtasks. Splitting into fewer numbers of tasks has two benefits:

1. Effective number of tasks in the Liu and Leyland's bound is reduced: $\Theta(n) = n(2^{\frac{1}{n}} - 1)$
2. It increases overall system utilization.

The remainder of this article is divided into seven sections: We firstly present our system model and notations in Section 2. In Section 3, related works are briefly reviewed. Section 4, describes the proposed RMLS semi-partitioned scheduling. Section 5, is the theoretical foundations and safeness proof of the algorithm; in Section 6, the algorithm is simulated and results are documented, and finally a summary and future work are presented in Section 7.

2. System Model and Notations

We consider a system with *m* symmetric processors and the main aim of this article is to schedule a bag-of-tasks containing periodic hard real-time tasks with an implicit deadline in such a system. A very important assumption is that all tasks are synchronous. That is, the deadline of any arbitrary task request is the exact time when the next request of that task arrives. We have also further assumed that all tasks are preemptive. From now on, in order to avoid potential difficulty in naming, we simplify call it *task* in this article.

The following notations are used throughout the article:

1. n: total number of tasks
2. n_1: total number of tasks and subtasks
3. m: total number of available or processors
4. m_1: total number of used processors
5. τ_i: i^{th} task
6. T_i: minimum time between any two consecutive requests of task τ_i, i.e. minimum request interval of τ_i
7. C_i: worst case computation time needed by every request of task τ_i. It is clear that $C_i \leq T_i$
8. $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$: set of all tasks (bag-of-task)
9. $\rho = \{P_1, P_2, ..., P_m\}$: set of all processors
10. u_i: the utilization of task τ_i which is equal to $u_i = \frac{C_i}{T_i}$
11. $U(\Gamma)$: total utilization of task-set Γ and subtasks

To evaluate the performance of scheduling heuristics, there are many theoretical factors such as utilization bound, speedup factor and many others. In addition to those factors, there are many articles which have used empirical methods to show and compare the relative performance of different algorithms. The most widely accepted are the number of randomly generated task-sets. Many different algorithms are now available to generate random task-sets and evaluate performance, but they are almost similar to one other in using parameters such as the number of total processors, task-set utilization, the number of all tasks in the task set, the distribution of task deadlines, the range of task periods, and the distribution of every task utilization. More details about the used performance evaluation algorithm in the present article and the number of randomly generated task-sets are provided in Section 6.

3. Related Works

Many studies have been conducted in the domain of homogeneous multiprocessor and multicore systems since the 1960s. Among those, semi-partitioned approaches, which try to merge and employ the best attributes of global and fully partitioned approaches, have a prominent role because of solving the task allocation problem which is analogous to the bin-packing problem. Anderson and Tovar [5], in one of the first studies in this respect, proposed an approach for scheduling hard real-time periodic tasks with implicit deadlines called EKG. In this approach, the parameter *k* has been used to control the splitting of tasks into the light and heavy sets. This suggestion is to set k=2 which yields the utilization bound of 66% and, on average, at most four preemptions per every task over the hyper period; yet, k=m gives the utilization bound of 100% and 2k preemption for each task. In this context, many researchers have proposed the semi-partitioned problem with Earliest Deadline First (EDF) scheduling [6], [7]. The best known worst-case utilization bound known using semi-partitioned EDF scheduling on multicores is 65% for

Earliest Deadline Deferrable Portion (EDDP) algorithm [8]. EDDP distinguishes between heavy tasks (those whose utilization are greater than 65%) and light tasks (other tasks with utilization < 65%) in such a way that the algorithm firstly assigns every heavy task to its own processor and then light tasks are placed on the remaining processors. They showed that 65% is a safe utilization bound for EDDP provided tasks which are periodic with implicit deadlines. Later, in 2009, they proposed EDF with Window-constraint Migration (EDF-WM) which has less context switching overhead [9].

On the other hand, relatively fewer algorithms are proposed for fixed-priority algorithms [10]. Rate Monotonic Deferrable Portion (RMDP) and Deadline Monotonic with Priority Migration (DM-PM) fixed-priority algorithms are proposed by Kato et al. [11, 12]. The worst-case utilization bound of those algorithms is shown to be 50%. RMDP consists of two phases: task assigning and task scheduling. The first phase is very simple: Sort all tasks in ascending order of T_i. Then, assign tasks to processors and if assigning a task causes the processor utilization bound to exceed, split the task into two subtasks. Sub-task 1 is placed on the current processor and sub-task 2 is assigned to the next processor. Task scheduling phase utilizes RM scheduler algorithm in every processor considering the fact that the second portion of a ready task in a processor cannot start to run until its first portion finishes execution.

PDMS_HPTS_DS is proposed by Lakshmanan et al. [2] which achieves the utilization bound of at least 60% providing tasks have an implicit deadline. It can, nevertheless, increase up to 65% if tasks are assigned to processors in order of decreasing utilization. Moreover, this bound can be extended to 69.3% for light tasks, i.e., tasks with utilizations less than 0.41. Guan et al. have proposed two algorithms which they called SPA1 and SPA2 [4], [10]. SPA2 has a pre-assignment phase in which special heavy tasks are first assigned to a separate processor. The advantages of this method are that the number of split tasks is m-1 and SPA2 reaches the worst-case utilization bound of 0.693. This is equal to Liu and Leyland's bound [13] for single processor systems The disadvantage includes, the worst-case bound in SPA2 is calculated using n which is the cardinality of the whole task-set, and every processor's utilization must be less than or equal to that. For example, although Liu and Leyland's least upper bound utilization for a two-task processor is approximately 0.83, but with SPA2 its utilization should not exceed 0.693. With this explanation, the claim that SPA2 has reached Liu and Leyland's utilization bound, does not seem to be entirely correct.

For supplementary information on hard real-time task scheduling algorithms and related issues, the reader is invited to refer to [14].

4. Rate Monotonic Least Splitting

The basic idea of the semi-partitioned method, which is being presented here, has been published in an in-progress research workshop [15]. In that paper, the fundamental theorem which shows the safeness of system was not proven. In addition, none of the other theoretical results provided by this article have appeared in that paper. Now, a brief introduction of the method is given here and new findings and performance evaluations follow. The method is called Rate-Monotonic Least Splitting (RMLS) because it is a semi-partitioned method in which at most m_1-1 tasks are split and at the same time, to the best of our knowledge, the method of partitioning and task splitting presented here is very simple, comprehensible and it is shown to be efficient.

Our experiments show that the achieved processor utilization is approximately 9.6% higher than the best-known results for general real-time systems, i.e., no restrictions on utilization of individual tasks, up to now.

Having taken all the above explanations into account, in the following sub-section, we first propose our novel idea to demonstrate how tasks are placed on processors. Then, in the next sub-sections, we will have a fairly good discussion on the computational complexity of the RMLS algorithm.

4.1. Task Assignment

The proposed assignment algorithm is precisely outlined in Fig. 1 and we elaborate on it in the two super steps as follows:

Step 1: Selection of single tasks and pairs of tasks to assign each one to a separate processor (lines 4 to 25).

Step 2: Assignment of remaining tasks to remaining processors (lines 28 to 47).

In the first step, tasks are sorted in descending order of their utilizations and the result is saved as a sorted task-set. A greedy approach is followed to find single tasks or pairs of tasks which can be assigned to separate processors to which other tasks will not be assigned to. To such processors, no subtasks will be assigned. In this phase, two pointers, i and j, are set to the beginning and the end of the set, respectively. If $\theta(3) \leq u_i + u_j \leq 1$ then these two tasks make a pair (lines 7 to 11) which is assigned to a separate processor; otherwise, one of these tasks is removed from the set (lines 18 to 22) and the process continues until the two pointers pass each other. The removed task will be scheduled in the second step. Step 1, continues by recognizing heavy tasks, i.e., a task τ_l with $u_l \geq \theta(2)$, and such task is assigned to a separate processor (lines 13 to 16.) No subtasks will be assigned to these processors.

To get the highest possible utilization, the scheduler of each processor with two tasks is a modified version of RM named Delayed Rate Monotonic (DRM) [16]. Suppose the two tasks $\tau_i = (C_i, T_i)$ and $\tau_j = (C_j, T_j)$, are solely assigned to processor p_k which has no other tasks. The delay time of each request of task τ_i, whose priority is higher than τ_j with respect to RM, is taken to be equal to T_i-C_i while there is no delay for requests of task τ_j. Here, you can consider that the delay is similar to (but not exactly the same as) ready time.

If the ready time of a request by τ_i is T_i-C_i and the request arrives at time t, then it would not be possible to start executing this request until time t+T_i-C_i. On the other hand, the delay of task τ_i is terminated at any time there is no pending request from task τ_j, and it will not re-enter delay state even if a new request arrives from τ_j. In addition, the delay of any request from this task can also end when T_i-C_i

time units have elapsed from the time that request is generated.

This modification will guarantee that if the total utilization of the two tasks τ_i and τ_j, assigned to a separate processor is less than or equal to one, then the processor will always run safely [17].

```
      Data: Task-set Γ //Includes Ti , Ci for each task i
      Result: Packing ρ
1     ρ←{}
2     k←0
3     Γ′←Γ
4     Sort Γ′ in descending order of tasks' utilization
5     i←1, j←length(Γ)
6     while i<j do
7       if Θ(3)≤u_τ'i + u_τ'j≤1 then
8         k←k+1
9         Add Pk to ρ
10        Move τ'i and τ'j from Γ to Pk
11        i←i+1 ; j←j-1
12      else
13        if Θ(2)≤u_τ'i then
14          k←k+1
15          Add Pk to ρ
16          Move τ'i from Γ to Pk
17        else
18          if Θ(3)≤u_τ'i + u_τ'j then
19            i←i+1
20          else
21            j←j-1
22          end
23        end
24      end
25    end
26    k←k+1
27    Add Pk to ρ
28    while Γ≠∅ do
29      τi←the task with the highest priority in Γ
30      if U(pk) = Θ(|pk|) then
31        k←k+1
32        Add Pk to ρ
33      end
34      if U(pk) + ui ≤ Θ(|pk| + 1) then
35        Move τi from Γ to Pk
36      else
37        if U(pk) < Θ(|pk| + 1) then
38          Select τj from Γ where
          u_τj + U(pk) <  Θ(|pk| + 2)
39          Move τj from Γ to Pk
40          Split τi into τi1 and τi2 such that
41            u_τi1 = Θ(|pk| + 1) − U(pk)
42          Replace τi in Γ by τi2 with u_τi2 = Cj2/(Tj−Cj1)
43          Move τi1 to Pk
44          k←k+1
45          Add Pk to ρ
46        end
47      end
48    end
49    m1←k      //number of used processors
```

Fig. 1. The packing algorithm

The scheduler of all processors, except those which has two tasks, is the traditional RM without any delay or ready time for requests.

Step 1, serves two sole purposes: (1) It increases the number of processors with higher utilization than those processors which are assigned tasks in Step 2, and (2) It increases the number of processors with no split task and hence, it decreases the total number of tasks which will be split in Step 2. Thus, by reducing the effective number of tasks (the total number of tasks and subtasks) the intuition is that there would be less number of tasks preemptions during run time.

In Step 2, all unassigned tasks will be sorted in decreasing order of RM priorities, i.e., the non-descending order of their request interval lengths. An empty processor is selected and then an unassigned task is selected from the sorted list to assign to the selected empty processor. This scenario will continue to repeat itself until the current task, say task τ_i, will overload the processor (lines 29 to 35). Then a search amongst the remaining unassigned tasks must be done to find a task with maximum utilization which can be assigned to this processor without overloading it. If one is found, it will be assigned to the processor. If this processor is not filled, task τ_i is then split into two subtasks so that the first subtask is assigned to the current processor and makes it full with respect to Liu and Layland's bound for the respective number of tasks and subtasks in this processor (lines 37 to 44).

To clarify, suppose that the current processor is p_k and task τ_i is the task which is split into two subtasks τ_{i1} and τ_{i2} with execution times C_{i1} and C_{i2}, respectively. The utilization of τ_{i1} is $u_{i1} = \frac{C_{i1}}{T_i}$ for processor p_k. A new processor, p_{k+1}, is picked up and the second part of task τ_i which was split, i.e., τ_{i2}, is assigned to this processor. Although the actual utilization of this subtask is $\frac{C_{i2}}{T_i}$, its effective utilization on processor p_{k+1} is taken to be:

$$u_{i2} = \frac{C_{i2}}{T_i - C_{i1}} \quad (1)$$

The effective utilization of this subtask is greater than its actual utilization, i.e. $\frac{C_{i2}}{T_i - C_{i1}} > \frac{C_{i2}}{T_i}$. Therefore, the difference of these two values is what we have to sacrifice because there may be some situations in which both processors that share task τ_i want to execute this task but the only processor that can run it at this time, is the processor whose index is the lower. In Lemma 3, we will prove that, in the worst case, the second part of a request from task τ_i will have a time length of T_i-C_{i1}, not T_i, to be executed. As mentioned earlier, this is due to the interference between the two processors that share task τ_i.

For example, suppose tasks τ_1= (1.1, 4), τ_2= (3, 17), and τ_3= (3.2, 18) are completely assigned to processor p_1 and task τ_4 = (6.55, 20) is broken into two subtasks $\tau_{4\text{-}1}$ = (2.55, 20) and τ_{4_2} = (4, 20) which are assigned to processors p_1 and p_2, respectively. Besides, tasks τ_5 = (5, 25) and τ_6 = (6, 30) are completely assigned to processor p_2 and from task τ_7 = (7, 42) the subtask τ_{7_1} = (5.65, 42) is assigned to processor p_2. The rest of task τ_7, i.e., τ_{7_2} = (1.35, 42) and task τ_8 = (47.4, 60) are assigned to processor p_3. The total utilization of these three processors are computed as follow:

$$U_1 = \frac{1.1}{4} + \frac{3}{17} + \frac{3.2}{18} + \frac{2.55}{20} = 0.7567.$$

$$U_2 = \frac{4}{20 - 2.55} + \frac{5}{25} + \frac{6}{30} + \frac{5.65}{42} = 0.7637.$$

$$U_3 = \frac{1.35}{42-5.65} + \frac{47.4}{60} = 0.8271$$

In RMLS algorithm outlined in Fig. 1, the process of assigning tasks to the processors continues until all tasks are assigned. If processors are exhausted but some unassigned tasks still remained, the assignment is unsuccessful; otherwise, it is successful.

Suppose the assignment is successful, RMLS splits at the most m_1-1 tasks, where m_1 is the actual number of used processors. There is no release time or delay time for the tasks that are assigned in Step 2 and thus, the scheduler is the traditional RM with a minor amendment. Obviously, it is clear that only one processor can execute a sequential task at any given time. In order to make sure this vital condition is observed, whenever there is a conflict, the processor with the lower index must have the precedence in executing the shared request. That is, under RMLS, the execution of that portion of the split task which is assigned to the lower indexed processor is not affected by the execution of that portion of the split task which is assigned to the higher index processor. However, the execution of that portion of the split task which is assigned to the higher indexed processor may be delayed because the lower indexed processor is running the split task. In other words, the lower indexed processor can run its own portion of the split task whenever it desires to, but the higher indexed processor can only run its own portion if the lowered indexed processor is not running its portion of the split task.

Using Equation (1) in computing the total utilization of processor p_{k+1} will reduce the actual sum of the task utilization on processor p_{k+1} to less than Liu & Layland's bound. However, this is an unavoidable cost that we have to pay for all processors to run safely.

On the positive side, by using RMLS, there is no need to restrict the sum of utilizations of all processors to be less than or equal to $\theta(n_1)$ (where n_1 is the total number of tasks and subtasks of the whole system after partitioning and complete assignment). That is, the total utilization could be more than 69.3% and the system is still in a safe state for any arbitrary number of processors (e.g. for a large number of processors).

4.2. Computational Complexity

To calculate computational complexity, we divide the scheduler into three steps. The task-set is sorted in the first step; thus, for a task-set of size n, the computational complexity of the first step would be O(nlog n). In the second step, a search for large tasks and pairs of tasks is done. This step is preceded by a loop. This loop continues until i and j variables, which respectively started from the beginning and the end of the tasks list, become equal. Since at least one of those variables is changed during each iteration, this loop iterates at most n times. Thus time complexity of the second step is to O(n). The third step assigns the remaining tasks (at most n tasks) to processors. This step contains a loop, which iterates one time for each task (either it is split or not). However, before every task splitting, a search must be conducted among unscheduled tasks. This search might find a small task to put in the current processor. As this procedure is done for every processor, the total time complexity of the third step would be O(mn), in which m is the number of processors used.

The total time complexity of the scheduling algorithm can be calculated by sum of complexities of steps 1 to 3, which is
O(max(n log n, mn)).

5. Safeness Verification of RMLS

The great advantage of RMLS is that Liu & Layland's bound is only computed based on the total number of tasks and subtasks assigned to every processor separately and that is why well-known similar previous researches could not reach such a high degree of freedom. As a fine example, Guan et al. [4] proposed a semi-partitioned algorithm with this additional constraint that the total utilization of all tasks coupled with subtasks must not exceed Liu & Layland's bound, whereas we successfully removed this restriction in our algorithm RMLS. We first present three lemmas and then prove the claimed statement.

In the rest of this article, it is assumed that two processors p_k and p_{k+1} share a task $\tau_i = (T_i, C_i)$ and for each request of the common task C_{i1} is executed by p_k and C_{i2} is executed by p_{k+1} so that $C_i=C_{i1}+C_{i2}$. In addition, effective utilization of the second part of a shared task is used as its utilization in the corresponding processor.

Lemma 1.

If Liu & Layland's bound, is satisfied by all processors, the second part of a request from a shared task, τ_i, between two processors, p_k and p_{k+1}, never overruns.

Proof.

The preference of executing a request from the shared task τ_i between processors p_k and p_{k+1} is given to p_k. Furthermore, the second part of a request from task τ_i has the highest priority within all tasks in p_{k+1}. Therefore, as soon as a request from task τ_i is generated, its execution starts by either p_k or p_{k+1} and continues executing (migrating between the processors, if necessary) until the second part of the task is completed. Therefore, in the worst case scenario, the execution of the second part of the task will be completed after a time length of C_i is passed from its request ($C_i \leq T_i$).

Definition 1.

A conflict-idle period is a time interval in which both processors, p_k and p_{k+1}, that share the shared task τ_i, want to run a request from the task, but because p_k is given a higher precedence, it will proceed with its execution; and at the same time, there is no other pending requests for processor p_{k+1} within this period and it will be idle. Note that, not all conflict periods of processors p_k and p_{k+1} are necessarily conflict-idle because if there are other requests for p_{k+1} then it will proceed with their execution and hence, it will therefore not be idle.

Consider a situation in which the task τ_i is split into subtasks τ_{i1} and τ_{i2}, and they are assigned to processors p_k and p_{k+1}, respectively. Subtask τ_{i1} is the task with the lowest priority (or in some cases the second lowest priority) within processor p_k while task τ_{i2} is always the task with the highest priority among all tasks and subtasks assigned to processor p_{k+1}. This decreases the chance of encountering a

situation in which both processors that want to simultaneously run the task τ_i; however, it is not zero. Therefore, conflict periods are very rare and as a result, seldom will conflict-idle periods to take place.

Lemma 2.

Suppose two processors p_k and p_{k+1} share a task τ_i and run n_k and n_{k+1} tasks while their total utilization is not greater than $\Theta(n_k)$ and $\Theta(n_{k+1})$, respectively. If there will not be any conflict-idle period with respect to τ_i, then both processors will always run overrun-free.

Proof.

Since processor p_k has a higher precedence to run τ_i than p_{k+1}, this processor will always run overrun-free. On the other hand, the only effect that p_k can have on the execution of tasks of processor p_{k+1} is that it may postpone the execution of the second part of a request from the shared task. This may harm the overrun-freeness of the shared task in p_{k+1} but it can be beneficial to the other tasks of this processor. However, in Lemma 1, it was proven that the second part of a request from a shared task never overruns. Therefore, this processor runs overrun-free as well.

Lemmas 1 and 2, will hold even if actual utilization of subtask τ_{i2}, i.e., $\frac{C_{i2}}{r_i}$, is used for computing the utilization of p_{k+1}. It is for compensation of possible conflict-idle periods that, in general, effective utilization of the shared task on processor p_{k+1} is computed as $\frac{C_{i2}}{T_i - C_{i1}}$.

Definition 2.

The remaining utilization of a request (not a task or subtask) at a given time is defined to be its remaining execution time divided by its remaining time to reach the deadline. At the exact time when a request is generated its remaining utilization is equal to its actual utilization. However, as time passes, its remaining utilization may fluctuate depending on how much time has been passed from its request time and how much it has been executed until the time that the remaining utilization is computed.

For example, suppose task $\tau = (10, 4)$ has generated a request at time 20 and the current time is 26 and up to now, this request has received 1.5 unit of CPU time. The remaining utilization of the request at time 26 is $(4 - 1.5)/(30 - 26) = 0.625$.

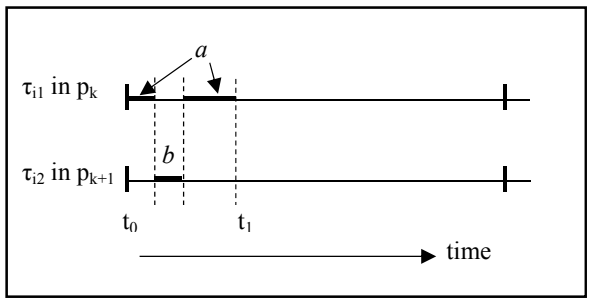

Fig. 2. A Sample execution of parts of a split task

Lemma 3.

Suppose two processors p_k and p_{k+1} share a task τ_i. The remaining utilization of a request from τ_i for processor p_{k+1} is maximal at the exact time when the execution of processor p_k's share of this request is completed when p_k starts this request immediately after it is generated and continues running it until its share is completed.

Proof.

Suppose as soon as a request from τ_i is generated at a time t_0, processor p_k starts to execute it until its share is finished at time t_0+C_{i1}. At this time, effective utilization of the subtask τ_{i2} on p_{k+1} is equal to $\frac{C_{i2}}{T_i - C_{i1}}$. We show that this is, in fact, the maximal effective utilization of τ_{i2}, which means subtask τ_{i2}'s effective utilization never becomes greater than this value. It is worth mentioning to recall that requests of task τ_i have the highest priority in processor p_{k+1}. This implies that any request for this task will be immediately picked up for execution by p_{k+1} if p_k is not executing it.

On the other hand, if the execution of the second part of a request of task τ_i is completed by processor p_{k+1} then its remaining utilization becomes zero and remains zero until a new request is generated from the same task.

With these points in mind, consider a situation where at any time t_1 ($t_0 \le t_1 \le t_0+C_i$), processor p_k has executed this request for the duration of length an ($a \le C_{i1}$), and processor p_{k+1} has executed the same request for duration b ($b < C_{i2}$ and $a+b = t_1-t_0$). This example is illustrated graphically in Fig. 2.

At time t_1 effective utilization of τ_{i2} is:

$$\frac{C_{i2} - b}{T_i - (a + b)}$$

Since $a \le C_{i1}$,

$$\frac{C_{i2}-b}{T_i-(a+b)} \le \frac{C_{i2}-b}{T_i-(C_{i1}+b)} = \frac{C_{i2}-b}{T_i-C_{i1}-b}$$

To show that the maximal effective utilization of τ_{i2} is $\frac{C_{i2}}{T_i-C_{i1}}$. it has to be shown that:

$$\frac{C_{i2}-b}{T_i-C_{i1}-b} \le \frac{C_{i2}}{T_i-C_{i1}}$$

That is,

$$(C_{i2} - b)(T_i - C_{i1}) \le C_{i2}(T_i - C_{i1} - b)$$

Or,

$$-bT_i + bC_{i1} \le -bC_{i2}$$

Or,

$$b(C_{i1} + C_{i2}) \le bT_i$$

Which is always true because b is positive and$C_{i1} + C_{i2} \le T_i$.

We have now provided a solid foundation of what must be considered to prove the safeness of multiprocessor systems to satisfy the requirements verbalized by RMLS scheduling algorithm.

Theorem 1.

If effective utilization of each of two processors p_k and p_{k+1} which share a task τ_i, is not greater than Liu and

Layland's bound, then both processors will always safely run their corresponding tasks and subtasks.

Proof.

This theorem is similar to Lemma 2 in which it is assumed that there will be no conflict-idle period. However, here, this restriction is removed. In Lemma 2, it is mentioned that processor p_{k+1} does not have any influence on the execution of tasks and subtasks assigned to processor p_k. Since Liu and Layland's bound is satisfied for p_k it will always safely run its assigned tasks and subtask. In the packing algorithm (Fig. 1), the utilization of the shared task on processor p_{k+1} is computed as $\frac{C_{i2}}{T_i - C_{i1}}$ which, based on Lemma 3 and is the maximum utilization that τ_{i2} can always impose on the processor p_{k+1}.

On the other hand, the utilization of processor is taken to be less than or equal Liu and Layland's bound. Therefore, this processor will always safely run its assigned tasks and subtask, as well.

In each processor p_k (k=1,2,3,.., m_1), at most, there is only one task which is shared with the processor p_{k-1} (if k>2) and one task which is shared with processor p_{k+1} (if $k<m_1$). Using Theorem 1 twice, once for p_{k-1} and p_k and once for p_k and p_{k+1}, we can conclude that all processors would be safe with RMLS.

6. Simulations

Despite the fact that theoretical results such as speedup factors, play a prominent role to verify the schedulability performance, over the last years, many different empirical studies have also aimed to investigate the relative schedulability test performance amongst many different scheduling algorithms in the domain of real-time systems. Empirical studies also provide more general schedulability tests by focusing more on individual tasks parameters which yield to take into account non-specific task-sets. Task parameters such as the number of processors and tasks, the total utilization of the task-set, task deadline distribution and period and many others, would be very important especially for those techniques that will be employed in safety-critical industrial environments.

In the current section, the proposed method is compared with SPA2 [10]. We used UUnifast algorithm, a de facto standard proposed by Bini and Buttazo [18], to produce random task-sets without any bias. It is given that the n tasks' utilization (as random variables) are uniformly distributed between 0 and 1, and the sum of them is a specific value which is the total utilization of the system. The UUnifast algorithm uses the cumulative distribution function and generates task-sets with uniform distribution in O(n) time order. For further reading please refer to [18], [19].

6.1. Comparisons

To compare the results of packing, we used the method used by Burns et al [7]. In this method, task-sets are divided into some categories. For each category of task-sets, the result mean of that category is selected for comparison purposes. Task-sets in each category have the same overall utilizations and the same number of tasks. For example, there are 200 task-sets in the first category, with their overall utilization is equal to 4, and for this category, there are 16 tasks in each task-set.

Experiments were performed on 3000 randomly generated task-sets with a different number of tasks and different overall utilizations. Overall utilizations used are 4, 8 and 16. The number of tasks tested with each utilization is shown in Figures 3 to 5. For example, we compared the three methods SPA2, RMLS, and RMLS+DRM with task-sets and with the overall utilization of 8 so that task-sets contain 16, 20, 28, 44, and 76 tasks.

We allow RMLS algorithm to assign processors as needed and, for the SPA2 algorithm, we initially start with a high number of processors with which we are assured of the safety of the system. Then, we gradually reduce the number of processors one at a time until reaching the lowest number of processors in which the system is still in a safe mode. When the minimum number of processors needed for each method is found, the average utilization of all processors is calculated by dividing the overall utilization of each task set by the number of processors simply.

The primitive version of RMLS (represented by PRMLS in Figures 3 to 5) uses RM scheduler in all processors. Thus, the utilization of systems containing two processors must not exceed the higher bound of θ (2). Moreover, in that version, single tasks whose utilizations are greater than or equal to 0.83, were not separated to schedule each one on a single processor. The partitioning algorithm of PRMLS is a simplified version of the algorithm outlined in Fig. 1, in which all activities concerning Step 1 is removed. We would now like to compare primitive RMLS with SPA2 and RMDP.

The median utilization of SPA2 is either the same or lower than PRMLS. However, for equal medians, error bars indicate that the efficiency of PRMLS partitioning is better than SPA2. The average utilizations achieved for whole task-sets are 0.680 and 0.735 for SPA2 and RMLS respectively. This shows that the overall performance of Primitive RMLS (PRMLS) is more than 8% higher than that of SPA2.

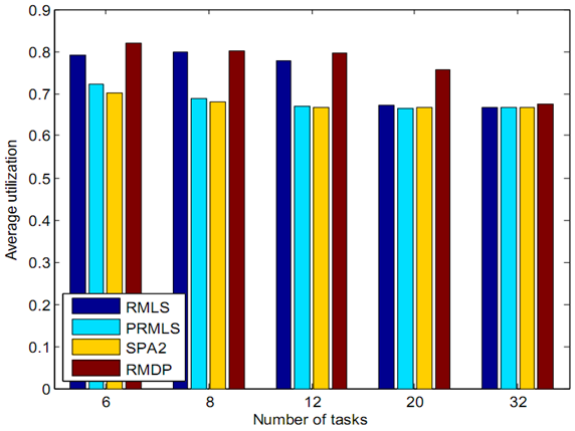

Fig. 3. Median of performance, by each method, for U=4

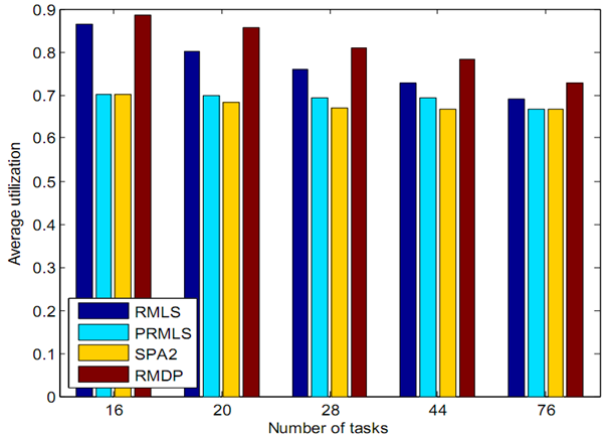

Fig. 4. Median of performance, by each method, for U=8

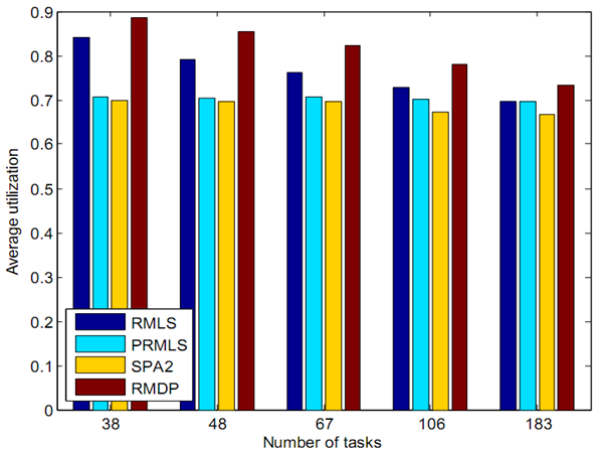

Fig. 5. Median of performance, by each method, for U=16

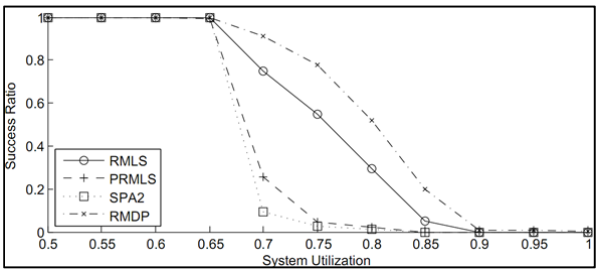

Fig. 6. Success rate for each method, for 3000 task sets

6.2. Discussion

Our experiments reveal that using DRM for two-task processors, greatly improvement the overall utilization of processors. The average utilization for RMLS on the randomly generated 3000 task-sets was 0.776. Medians, 25 and 75 percentiles are shown in Figures 3 to 5. An overall improvement of more than 14% as compared to SPA2 is a remarkable achievement for RMLS.

By comparing RMLS and PRMLS, one gets the impression that a little change can have a great performance improvement (i.e. close to 6%). Experiments, however, show better performance for RMDP method, which is caused by its scheduler. The scheduler of the RMDP method differs from rate-monotonic and, in actual fact, it is really more complex and involved; thus, its better performance is expected.

Although one can infer from Figures 3 to 5, that SPA2 usually should use a higher number of processors to safely schedule the same set of tasks as compared to both of PRMLS and RMLS, some complement charts are provided for visual comparisons (see Figures 7 to 9).

In addition, another experiment was conducted to test the schedulability of RMLS, PRMLS, SPA2, and RMDP on total 3000 randomly generated task-sets (see Fig. 6). In this experiment, we set different system utilizations and measure the ratio of task-sets that are schedulable. Taking a quick look at Figure 6, we clearly see that for all task-set with total utilization $U(\Gamma) <= 0.66$, all four algorithms can schedule every task-set with success ratio 1; however, for task-sets with total utilization greater than 0.66, SPA2 and PRMLS show their downside so that system total utilization of below 0.7, for instance, only 9% of randomly generated task-sets are schedulable using SPA2 whereas RMLS schedules about 79% of task-sets safely.

As was mentioned before, the high ability of RMDP to schedule task-sets is owing to its complicated algorithm. Additionally, please note that RMDP scheduler policy does not rate-monotonic.

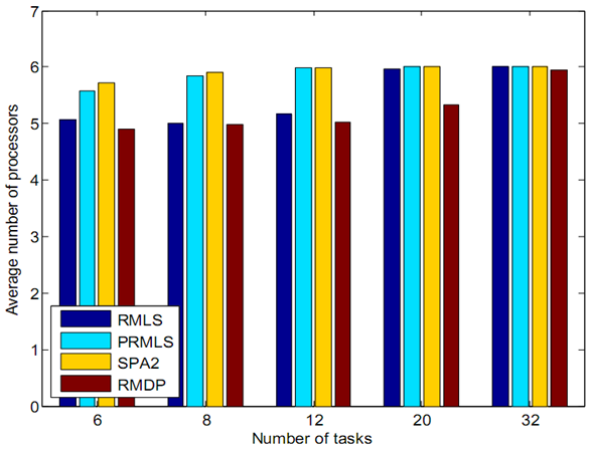

Fig. 7. Number of used processors for each method, for U=4

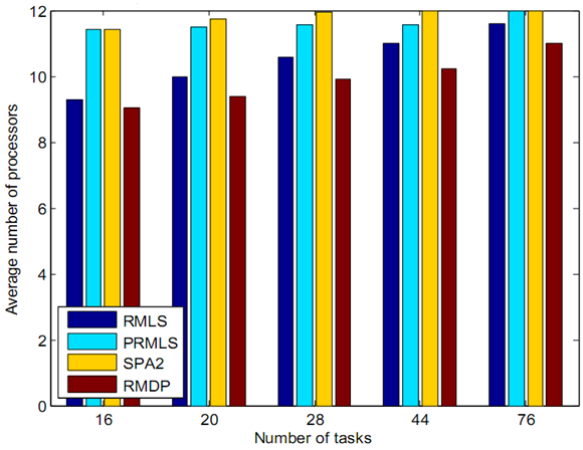

Fig. 8. Number of processors used for each method, for U=8

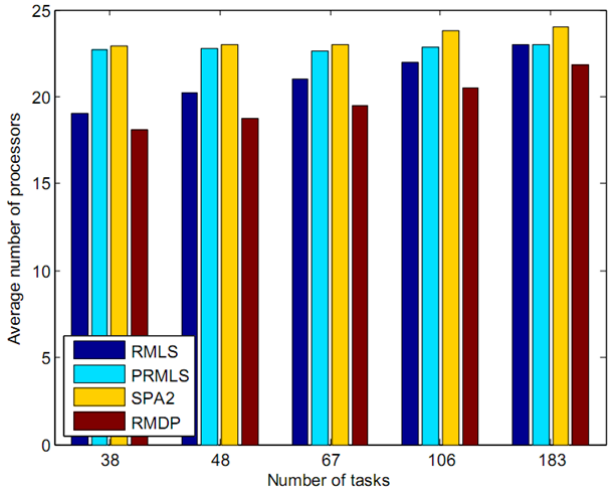

Fig. 9. Number of processors used for each method, for U=16

7. Summary and Future Work

Significant advances which have been made in many industrial areas are good evidence to support this claim that the importance of utilizing embedded multiprocessor system-on-a-chip (MPSoC) is an undeniable fact. In this context, regardless of many problems that must be considered, task management is a key issue which we focused on in the current article. Out of different approaches for hard real-time task scheduling, semi-partitioning of periodic tasks on multiprocessors was studied here in which the scheduler of each processor is rate monotonic, with the exception that the scheduler of processors with exactly two whole tasks is delayed rate monotonic (DRM) [16]. It was proven that when a task is split between two processors, if the utilization of the second part of the task is considered a little higher than its actual utilization, and RM scheduling policy is used on both processors while Liu and Layland's bound is satisfied, both processors run safely and all tasks meet their deadlines.

With this method, there is no need to define a release time for the second subtask. The Rate-Monotonic Least Splitting (RMLS) algorithm was developed and its performance was compared with the SPA2 algorithm as well as RMDP. It was concluded that the performance of PRMLS (Primitive RMLS) is more than 8% higher than SPA2 and the performance of RMLS is more than 14% higher than that of SPA2. This means that both PRMLS and RMLS usually need a fewer number of processors to safely schedule the same set of real-time tasks than SPA2, using a semi-rate-monotonic scheduler.

Although many types of research in the domain of semi-partitioned scheduling are being conducted, the use of new methods seems to be very important to improve real-time scheduling performance on multiprocessors. For example, authors in [20] did employ the equation of the line to dynamically assign priority to the tasks (called LTS) which appear to be an interestingly novel method in global multiprocessor scheduling. They claimed that their algorithm schedules all periodic task sets with total utilization up to 100% safely. One can be to modify (and improve) the LTS algorithm so that it is possible to use in semi-partitioned multiprocessor scheduling (e.g. the policy of each processor scheduler).

8. Acknowledgment

The authors wish to sincerely acknowledge the advice and support that they have received from Professor Mahmoud Naghibzadeh, the director of Knowledge Engineering Research Group (KERG) at Ferdowsi University of Mashhad. He generously provided insight and expertise that greatly assisted the research, even though he did not completely agree with all of the conclusions and interpretations of the current article.

References

[1] C.L. Liu, "Scheduling algorithms for multiprocessors in a hard real-time environment". *JPL Space Programs Summary,* vol. 37-60, 1969, pp. 28–31.

[2] K. Lakshmanan, R. Rajkumar and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors", *in Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on. IEEE*, pp. 239–248, 2009.

[3] M.R. Gary and D.S. Johnson: "Computers and Intractability; A Guide to the Theory of NP-Completeness" (W. H. Freeman & Co.), 1979.

[4] N. Guan, M. Stigge, W. Yi and G. Yu, "Fixed-priority multiprocessor scheduling with liu and layland's utilization bound", *in Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010 16th IEEE. IEEE, pp. 165–174, 2010.

[5] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions", in Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on. IEEE, pp. 322–334, 2006.

[6] J. Anderson, V. Bud and U. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems", *in Real-Time Systems.(ECRTS 2005). Proceedings. 17th Euromicro Conference on, 2005,* pp. 199–208, 2005.

[7] Burns, R. I. Davis, P. Wang and F. Zhang, "Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme", *Real-Time Systems,* vol. 48, no. 1, pp. 3–33, 2012.

[8] S. Kato and N. Yamasaki, "Portioned edf-based scheduling on multiprocessors", *in Proceedings of the 8th ACM international conference on Embedded software. ACM,* 2008, pp. 139–148.

[9] S. Kato, N. Yamasaki and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors", *in Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on.* IEEE, pp. 249–258, 2009.

[10] N. Guan and W. Yi, "Fixed-priority multiprocessor scheduling: Critical instant, response time and utilization bound", *in Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. IEEE*, pp. 2470–2473, 2012.

[11] S. Kato and N. Yamasaki, "Portioned static-priority scheduling on multiprocessors", *in Parallel and*

Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. IEEE, pp. 1–12, 2008.

[12] S. Kato and N. Yamasaki, "Semi-partitioned fixed-priority scheduling on multiprocessors", *in Real-Time and Embedded Technology and Applications Symposium,* 2009. RTAS 2009. 15th IEEE. IEEE, pp. 23–32, 2009.

[13] L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", *Journal of the ACM (JACM)*, vol. 20, no. 1, 1973, pp. 46–61.

[14] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems", *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[15] M. Naghibzadeh, P. Neamatollahi, R. Ramezani, A. Rezaeian and T. Dehghani, "Efficient semi-partitioning and rate-monotonic scheduling hard real-time tasks on multi-core systems", *in Industrial Embedded Systems (SIES), 2018 8th IEEE International Symposium on. IEEE,* 2013, pp. 85–88.

[16] M. Naghibzadeh, "A modified version of rate-monotonic scheduling algorithm and its' efficiency assessment", *in Object-Oriented Real-Time Dependable Systems, 2002. (WORDS 2002). Proceedings of the Seventh International Workshop on,* 2002, pp. 289-294.

[17] M. Naghibzadeh and K.H. Kim "The yielding-first rate-monotonic scheduling approach and its efficiency assessment", *International Journal of Computer System Science & Engineering*, pp. 173-180*,* 2003.

[18] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests", *Real-Time Systems,* vol. 30, no. 1-2, pp. 129–154, 2005.

[19] R. I. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems", *Real-Time Systems,* vol. 47, pp. 1-40, 2011.

[20] Ghavidel, M. Hajibegloo, A. Savadi and Y. Sedaghat, "LTS: Linear task scheduling on multiprocessor through equation of the line", *in Computer Architecture and Digital Systems (CADS),* 2015 18th CSI International Symposium on, pp. 1-6, 2015.